NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

MACHINE LEARNING FEATURE SELECTION FOR TUNING MEMORY PAGE SWAPPING

by

Rick Battle

September 2013

Thesis Advisor:	Craig Martell
Second Reader:	Joel Young

REPORT DOCUMENTATION PAGE						*Form Approved* *OMB No. 0704–0188*
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704–0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**						
1. REPORT DATE *(DD–MM–YYYY)* 23–9–2013		**2. REPORT TYPE** Master's Thesis				**3. DATES COVERED** *(From — To)* 2012-06-01—2013-09-27
4. TITLE AND SUBTITLE Machine Learning Feature Selection for Tuning Memory Page Swapping						**5a. CONTRACT NUMBER**
						5b. GRANT NUMBER
						5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Rick Battle						**5d. PROJECT NUMBER**
						5e. TASK NUMBER
						5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943						**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of the Navy						**10. SPONSOR/MONITOR'S ACRONYM(S)**
						11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited						
13. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number XXX.						
14. ABSTRACT						
15. SUBJECT TERMS Linux Kernel, Virtual Memory, Machine Learning, Temporal Locality						
16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**		**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**				**19b. TELEPHONE NUMBER** *(include area code)*
Unclassified	Unclassified	Unclassified	UU	55		

This thesis is an exploration of the virtual memory subsystem in the modern Linux kernel. It applies machine learning to find areas where better page-out decisions can be made. Two areas of possible improvement are identified and analyzed. The first area explored arises because pages in a computation appear repeatedly in a sequence. This is an example of temporal locality. In this instance, we can predict pages that will not be recalled again from the backing store with a precision and recall of 0.82 and 0.81, respectively, with a baseline of 0.30. The second is trying to predict when the system has made bad page-out decisions, those which lived in the backing store for less than one second before being recalled into RAM. In this case, we achieved a precision of 0.82 and a recall of 0.81 with a baseline of 0.12.

THIS PAGE INTENTIONALLY LEFT BLANK

MACHINE LEARNING FEATURE SELECTION FOR TUNING MEMORY PAGE SWAPPING

Rick Battle
Lieutenant, United States Navy
B.S., Virginia Tech, 2009

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2013

Author:	Rick Battle
Approved by:	Craig Martell Thesis Advisor
	Joel Young Second Reader
	Peter Denning Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis is an exploration of the virtual memory subsystem in the modern Linux kernel. It applies machine learning to find areas where better page-out decisions can be made. Two areas of possible improvement are identified and analyzed. The first area explored arises because pages in a computation appear repeatedly in a sequence. This is an example of temporal locality. In this instance, we can predict pages that will not be recalled again from the backing store with a precision and recall of 0.82 and 0.81, respectively, with a baseline of 0.30. The second is trying to predict when the system has made bad page-out decisions, those which lived in the backing store for less than one second before being recalled into RAM. In this case, we achieved a precision of 0.82 and a recall of 0.81 with a baseline of 0.12.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgements

I would like to thank everybody who helped me with this research:

Craig Martell
Joel Young
Stefan Lippers-Hollmann
Alric Althoff
Pranav Anand

And of course, my parents and girlfriend, for their unending support throughout all of this.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction

Computer systems have a hierarchy of memory that is traditionally broken up into three primary tiers. The first tier is the local cache available to a CPU. The second tier is the system's RAM. The third tier is the backing store, which today is generally either a hard drive or a solid state drive. In early computers, programs and data had to fit inside of the available RAM. To allow for larger than RAM problems, virtual memory was developed, where processes are given an address space larger than the physically available RAM and it is left to the system to decide which physical pages are kept in RAM and which are in the backing store.

Moving data between the first and second memory tiers is handled by the cache replacement algorithm implemented on the CPU. When workloads expand beyond the size of available RAM, decisions must be made about what to move down to the third tier since the difference in access times between RAM and the backing store is several orders of magnitude. These decisions are made by the page replacement algorithm (PRA) of the operating system. The difference between a good PRA and a bad one is the difference between a usable system under heavy load that eventually completes its task and a system lost to endless thrashing.

Machine learning was originally conceived with the notion of giving computers human-like intelligence and reasoning abilities. Today the field, while not achieving human-like abilities, is a well developed, nearly to the point of commoditization, array of classification, identification, and regression algorithms. Though there will always be room for new algorithmic developments, most research in the field, more commonly referred to as data mining now, is about data transformation for input into the preexisting models and assessment of results in new applications.

1.1 Thesis Structure

The research presented in this thesis seeks to apply modern machine learn techniques to data gathered from the virtual memory subsystem of the Linux Kernel, searching for inefficiencies and make recommendations for optimization.

In Chapter 2, we will present a brief history of virtual memory, explain the principle of locality and demonstrate its applicability to modern systems, explain working set theory, explore many common paging algorithms along with some of their advantages and pitfalls, explore the machine learning algorithms used in this thesis, and present the machine learning software package Orange.

In Chapter 3, we will present the hypothesis for this research, explain the experimental setup for data gathering along with some limitations of our approach, and the call-all-X method for calculating baseline, which will be used extensively in Chapter 4.

In Chapter 4, we will present the findings from various methods we applied to split and transform the data. In most cases were not able to do well, but in two important cases we did very well. In the first case, when the prior faulting page was the same as the prior-prior faulting page (see Figure 3.4 for a visualization) we show success in classifying those pages that would not be recalled from the backing store. In the second case, we will show success in classifying those pages that lived in the backing store for less than one second, making them very bad page out decisions.

In Chapter 5, we will explain the major contributions from this thesis, namely, a method of modifying the Linux kernel for data collection about page-out events and how the method can be used to build a data set useful for input into machine learning algorithms. From the data set, we will discover two cases in which improvements to Linux's page replacement algorithm can be made as discussed in the previous paragraph. Further analysis of the two cases in which we saw success will be performed and possible future work will be discussed.

CHAPTER 2: A Brief History of Paging and Machine Learning

The problem of developing efficient paging algorithms first began to be seriously studied in the 1960's: László Bélády, working for IBM, published a comprehensive study of paging algorithms, in which he laid out the fundamentals of the field. "To minimize the number of replacements, we attempt to first replace those blocks that have the lowest probability of being used again. Conversely, we try to retain those blocks that have a good likelihood of being used again in the near future." [1] He also set the stage for the problems of theoretically minimal algorithms to follow, "A good algorithm is one that strikes a balance between the simplicity of randomness and the complexity inherent in cumulative information. In some cases, too much reliance on cumulative information actually resulted in lower efficiency." [1]

2.1 Virtual Memory

Except for rare cases in history, such as the transition from 32bit to 64bit architectures when most systems came with 3GB of RAM, the physical amount of RAM present in a system is smaller than the limit of the address space of an operating system. This problem is exacerbated when running in a multiprogramming environment. Virtual memory is a solution to this problem and is built into nearly every operating system today. Virtual memory assigns the entire address space to each process, so, to the application it appears as though it is the only thing running. The memory assigned to the process has to be translated from physical addresses to the virtual addresses used inside the program. Since each program is able to address the entirety of the address space, it is quite common for the set of running applications to allocate more memory than is physically present in the system. When this occurs, the system must decide what to keep in physical RAM and what to send down to the backing store to make room for the newly requested memory. This process is called *paging*. The decision of what to keep and what to evict from RAM is the subject of this research.

2.1.1 Principle of Locality

Bélády is responsible for what is widely considered to be a fundamental theory of paging, the *Principle of Locality*. Bélády describes four kind of locality: temporal, spatial, branch, and equidistant locality. Temporal locality is the proposition that if a memory location was accessed recently, it is likely to be used again in the near future. Spatial locality is the proposition that if a memory location was used recently that memory locations near it are likely to be used in the near future. Branch locality has to do with the nature of `if-then` statements common to computer programs. Though the `else` block of a particular `if-then` segment may not be "close" to the `if` segment, it can still be likely to be accessed in the near future. Equidistant locality is a special case combination of spatial and temporal locality. If a segment of code exhibits both, it can easily be predicted which memory address will be accessed next.

The principle of locality was first proposed by Bélády in 1966 and articulated by collaborative work between Bélády and Denning in 1968. As such, it is occasionally questioned as to whether the principle of locality still applies to modern computer programs. As recently as 2011, spatial and temporal locality were both questioned and shown conclusively to still hold [4]. Instead of copying their graphs for inclusion here, we duplicated their results for demonstrative purposes, which can be seen in the memory access trace of a session of xterm in Figure 2.1.

2.1.2 Working Set

From the Principle of Locality, Peter Denning developed the notion of a program's working set [2]. The working set of an application is the set of all pages required by a program to operate correctly in a given time slice. This is a very powerful theory, as it allows operating system and, more specifically, PRA designers to make the correct decisions about what pages to evict during a page fault. If a page is not part of a program's working set, then it is the perfect candidate for eviction, no matter how frequently or recently it's been accessed [2]. From the data gathered while duplicating the results for verification of working set, Figure 2.1 shows the working set of xterm for any given time slice to be only those memory addresses shown in the graph. As can be seen, the working set grows and shrinks over time, but has clear blocks with no changes and significant periods of time where previously accessed memory locations are not needed for the program's current operation.

2.1.3 Paging Algorithms

There are many algorithms for deciding which page to discard from main memory when the system is experiencing memory pressure. The optimum algorithm is unfortunately impossible to implement in the general case, as it requires knowledge of the future. The optimum page replacement algorithm pages out the page whose next use is furthest in the future [1]. The best an implementable page replacement algorithm can hope to do is closely approximate the optimum algorithm.

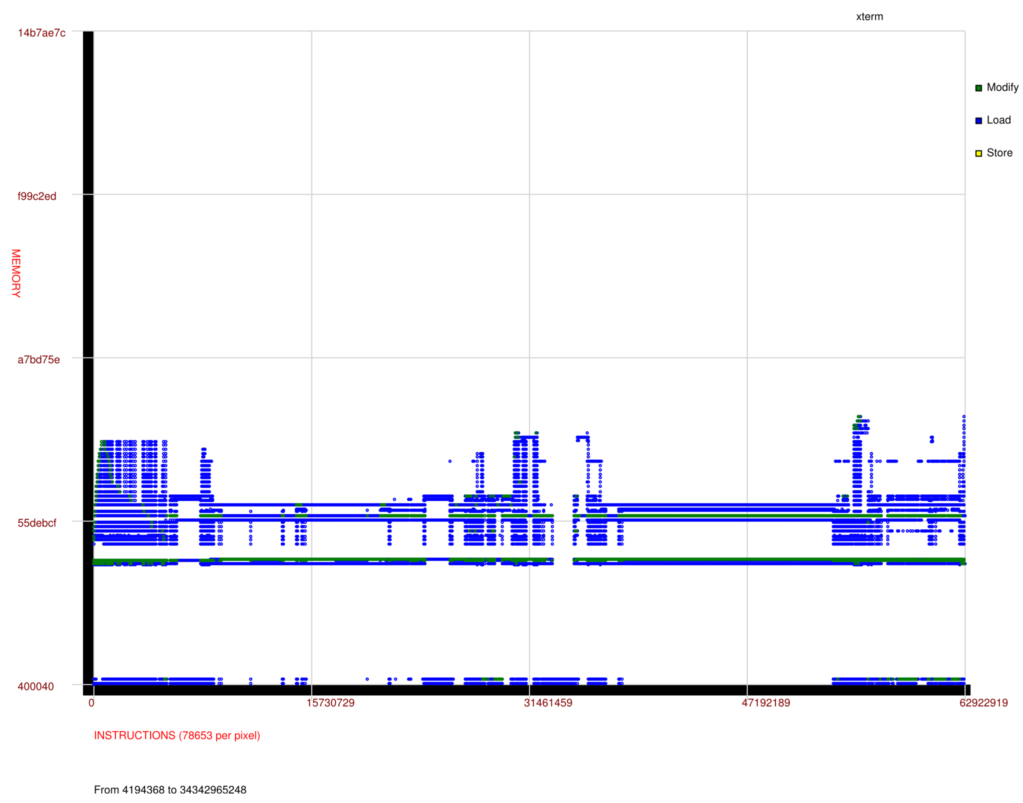


Figure 2.1: Visualization of the memory accesses pattern of a session of xterm clearly demonstrating spatial and temporal locality. Note the clusters in the graph for a given time slice, that is xterm's working set at that time.

The simplest page replacement algorithm is *random selection*. It assumes that every page is equally likely to be used again and selects a page at random to be moved to the backing store. This assumption is erroneous and generally results in useful pages being paged out too early, only to be paged back in shortly there after. [1]

The *first in/first out* (FIFO) page replacement algorithm assumes that the page which has been in RAM the longest is the least likely to be used again and thus sent down to the backing store. Pages are kept in a queue. When a page is allocated, it is placed at the head of the queue. When pages need to be paged out, the pages at the tail of the queue are selected. This algorithm has been shown to have significant shortcomings. When using a FIFO PRA, it is possible to encounter a situation in which decreasing the size of the available RAM can cause a decrease in running time, i.e., fewer page faults [3]. Imagine a loop which indexes an array one page longer than the length of the queue. After a first pass through the loop, the next page to be accessed will always be that page which was just paged out. FIFO is therefore, a sub-optimal page replacement algorithm.

Least recently used (LRU) is incredibly simple in concept, but has proven quite difficult to implement efficiently, the idea being that the page which was accessed the longest time ago is the best candidate for eviction [9]. The trouble comes with having to perform some combination of store, search, and sort on the list of evictable pages. Searching and sorting are computationally expensive operations and thus take too long to execute in a page fault handler. Moving the list management code out of the page fault handler solves the time requirement for the interrupt handler, but the overhead of managing a priority queue has been shown to have significant performance implications. The operations are sufficiently expensive that operating a true LRU resulted in degraded system performance when compared to simpler algorithms, unless running on a system with the required access time tracking hardware built in, as is done on some mainframes.

Second chance is not itself a page replacement algorithm, but a modification to an algorithm. In general, second chance can be applied to any PRA, but is most commonly used with FIFO and LRU. When a page that is a candidate for eviction is the target of the PRA to get evicted, the page gets passed over the first time instead of being paged out. If the PRA makes it through the entire list of evictable pages without evicting any or enough pages, then on the second pass the pages that were previously passed over will be moved down to the backing store.

Least recently used with exemptions (LRUe) is a modification to LRU where local context matching is used to build a short list of pages which are exempted from the normal paging process. Local context matching is historical tracking of page-use sequences. If pages B through D were used shortly after a page A recently in the program history, then pages B through D should not be a candidate for eviction during a page fault if the page A was used recently. In certain use cases, LRUe frequently outperforms standard LRU for the right load. [10]

Least recently used two queue (LRU 2Q) breaks the LRU list into two queues one holding frequently accessed pages the other containing infrequently accessed pages. LRU 2Q has a constant time overhead, out performs LRU, and requires no tuning. [11]

Adaptive replacement cache (ARC) is a self-tuning, low overhead replacement cache. [5] It was developed and patented by IBM. ARC outperforms LRU and the algorithm is used in several production IBM storage systems. Due to the algorithm being patented, it is highly unlikely that it will be used in open source software such as the Linux kernel prior to the patent expiring.

Clock is similar to FIFO and outperforms second chance except its list is circular, i.e., no push/pop operations are required. When pages are first added to the list, their use bit is set to one. Clock scans the list of evictable pages starting where it last left off, searching for an eviction candidate, i.e., a page whose use bit is zero. If the page being inspected's use bit is a one, it sets it to zero. When it finds a page who's use bit is zero, it checks the dirty bit. If the dirty bit is one, it schedules that page to be cleaned otherwise, it evicts that page. [7]

WSClock is a slight modification of clock. During the sweep, it checks for cleared use bits just as clock does. When it encounters a page who's use bit is cleared, it checks the time of last use field and if it's greater than the length of a time slice. If it is greater then the page is not part of the program's working set. It's dirty bit is then checked and handled the same way as in clock. [8]

Clock with adaptive replacement (CAR) is another self-tuning page replacement algorithm developed and patented by IBM. It maintains two clocks based on recency and frequency of use, backed by LRU lists. CAR out performs ARC [6], but again due to patent concerns in ineligible for use in open source projects such as the Linux kernel.

The virtual memory subsystem of the Linux kernel makes use of a simplified LRU 2Q algorithm for its page frame reclamation policy [12]. It's simplified in that instead of maintaining true LRU lists or FIFO lists, the two lists operate in a clock-like fashion. As such, the actual placement of a page inside a list is meaningless.

2.2 Machine Learning Algorithms

There are many machine learning algorithms available today. For this research, we chose to focus on supervised learning techniques, in particular, support vector machines, random forests, naïve bayes, k-nearest neighbors, and CN2.

The *support vector machine* (SVM) algorithm is a supervised machine learning technique useful for both classification and regression. The classifications made by an SVM are non-probabilistic binary and linear. The feature space may be transformed by polynomial, radial basis, sigmoid, or many other kernels. Data must be collected from some source, put into a feature vector, and labeled as being of a class or not of a class. The labeled data is then fed into the SVM, where it finds a hyperplane that divides the feature space [13].

Random forests are a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. [14] It can more easily be thought of as a sequence of decision trees. Because of the law of large numbers, random forests tend to be less susceptible to noise and thus are less likely to overfit [14].

Naïve bayes, based on Bayes' theorem with strong independence assumptions, is a probabilistic classification technique. Despite the fact that the independence assumptions are generally inaccurate in the real world, the classifier is highly effective and far less computationally expensive, especially for extremely high dimensional data sets. [15]

K-nearest neighbors is a non-parametric classification technique based on clustering of data points inside the feature space. It is one of the simplest machine learning techniques. A particular data point is classified by a majority vote of the *k*, typically a small number dependent on the amount of noise in the training data, nearest other data points. [16]

The *CN2* algorithm provides efficient induction of simple, comprehensible production rules in domains where problems of poor description language and/or noise may be present. [17]

2.3 Chapter Summary

In this chapter we presented a brief history of virtual memory, explained the principle of locality and demonstrated its applicability to modern system, explained working set theory, explored many common paging algorithms along with some of their advantages and pitfalls, and explored the machine learning algorithms used in this thesis. In the next chapter we will explain how we gathered the data and transformed it for input into the machine learning software package Orange and how we calculated baseline.

CHAPTER 3: Data Collection

Our hypothesis is that a machine learning based virtual memory (MLVM) system's page replacement algorithm will produce fewer page faults than the LRU2Q found in the current Linux kernel. In this chapter, we present the infrastructure needed to test this hypothesis.

3.1 Testing Environment

Tests were conducted on a virtual machine with artificially induced heavy system load.

3.1.1 Virtual Machine

All tests were conducted on a kernel based virtual machine (KVM) running a modified version of Aptosid 2013-01, a Debian derivative based on version 3.8 of the Linux kernel, running on a custom built desktop.

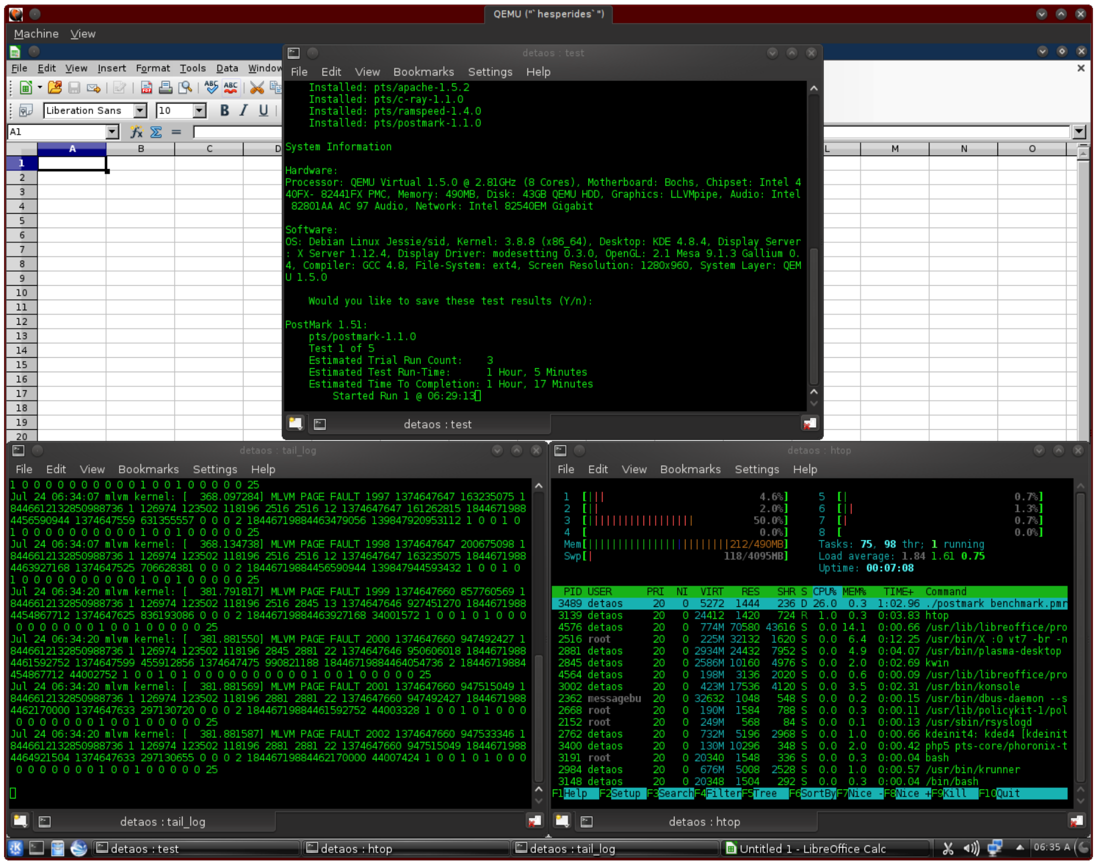
Motherboard: Asus P7P55D
Processor: Intel Core i7-860
RAM: G.Skill 12800
SSD: Pair of Samsung 840's running in RAID1

3.1.2 Inducing Heavy System Load

Heavy system load wad induced with the Phoronix Test Suite's Complex System Test. Memory pressure was induced with a bash script, induce_memory_pressure (see Appendix B.3), which alternatively opens a web browser with a random selection of websites in multiple tabs or opens a randomly selected office application. The browser or office application was closed prior to deciding which to open next. To ensure a significant number of page faults, the VM was limited to 512MB of RAM. See Figure 3.1 for a screenshot of the VM running.

3.2 Instrumenting the Kernel

To collect the data, modification were made to the virtual memory subsystem of the stock Linux kernel, which allowed us to better understand and analyze the inner workings of Linux's current page replacement algorithm.



```
    Installed: pts/apache-1.5.2
    Installed: pts/c-ray-1.1.0
    Installed: pts/ramspeed-1.4.0
    Installed: pts/postmark-1.1.0

System Information

Hardware:
Processor: QEMU Virtual 1.5.0 @ 2.81GHz (8 Cores), Motherboard: Bochs, Chipset: Intel 4
40FX- 82441FX PMC, Memory: 490MB, Disk: 43GB QEMU HDD, Graphics: LLVMpipe, Audio: Intel
 82801AA AC 97 Audio, Network: Intel 82540EM Gigabit

Software:
OS: Debian Linux Jessie/sid, Kernel: 3.8.8 (x86_64), Desktop: KDE 4.8.4, Display Server
: X Server 1.12.4, Display Driver: modesetting 0.3.0, OpenGL: 2.1 Mesa 9.1.3 Gallium 0.
4, Compiler: GCC 4.8, File-System: ext4, Screen Resolution: 1280x960, System Layer: QEM
U 1.5.0

    Would you like to save these test results (Y/n):

PostMark 1.51:
    pts/postmark-1.1.0
    Test 1 of 5
    Estimated Trial Run Count:    3
    Estimated Test Run-Time:      1 Hour, 5 Minutes
    Estimated Time To Completion: 1 Hour, 17 Minutes
        Started Run 1 @ 06:29:13
```

```
1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 25
Jul 24 06:34:07 mlvm kernel: [  368.097284] MLVM PAGE FAULT 1997 1374647647 163235075 1
8446612132850988736 1 126974 123502 118196 2516 2516 12 1374647647 161262815 1844671988
4456590944 1374647559 631355557 0 0 0 2 18446719884463479056 139847920953112 1 0 0 1 0
1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 25
Jul 24 06:34:07 mlvm kernel: [  368.134738] MLVM PAGE FAULT 1998 1374647647 200675098 1
8446612132850988736 1 126974 123502 118196 2516 2516 12 1374647647 163235075 1844671988
4463927168 1374647525 706628381 0 0 0 2 18446719884456590944 139847944593432 1 0 0 1 0
1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 25
Jul 24 06:34:20 mlvm kernel: [  381.791817] MLVM PAGE FAULT 1999 1374647660 857760569 1
8446612132850988736 1 126974 123502 118196 2516 2845 13 1374647646 927451270 1844671988
4454867712 1374647625 836193086 0 0 0 2 18446719884463927168 34001572 1 0 0 1 0 1 0 0 0
 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 25
Jul 24 06:34:20 mlvm kernel: [  381.881550] MLVM PAGE FAULT 2000 1374647660 947492427 1
8446612132850988736 1 126974 123502 118196 2845 2881 22 1374647646 950606018 1844671988
4461592752 1374647599 455912856 1374647475 990821188 18446719884464054736 2 18446719884
454867712 44002752 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 25
Jul 24 06:34:20 mlvm kernel: [  381.881569] MLVM PAGE FAULT 2001 1374647660 947515049 1
8446612132850988736 1 126974 123502 118196 2881 2881 22 1374647660 947492427 1844671988
4462170000 1374647633 297130720 0 0 0 2 18446719884461592752 44003328 1 0 0 1 0 1 0 0 0
 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 25
Jul 24 06:34:20 mlvm kernel: [  381.881587] MLVM PAGE FAULT 2002 1374647660 947533346 1
8446612132850988736 1 126974 123502 118196 2881 2881 22 1374647660 947515049 1844671988
4464921504 1374647633 297130655 0 0 0 2 18446719884462170000 44007424 1 0 0 1 0 1 0 0 0
 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 25
```

```
  1  [|||                   4.6%]     5  [|                     0.7%]
  2  [||                    2.0%]     6  [||                    1.3%]
  3  [|||||||||||||||||    50.0%]     7  [|                     0.7%]
  4  [                      0.0%]     8  [                      0.0%]
  Mem[||||||||||||||||||||||||212/490MB]     Tasks: 75, 98 thr; 1 running
  Swp[|                118/4095MB]     Load average: 1.84 1.61 0.75
                                       Uptime: 00:07:08

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 3489 detaos     20   0  5272  1444   236 D 26.0  0.3  1:02.96 ./postmark benchmark.pmr
 3139 detaos     20   0 24412  1420   724 R  1.0  0.3  0:03.83 htop
 4576 detaos     20   0  774M 70580 43616 S  0.0 14.1  0:00.66 /usr/lib/libreoffice/pro
 2516 root       20   0  225M 32132  1620 S  0.0  6.4  0:12.25 /usr/bin/X :0 vt7 -br -n
 2881 detaos     20   0 2934M 24432  7952 S  0.0  4.9  0:04.07 /usr/bin/plasma-desktop
 2845 detaos     20   0 2586M 10160  4976 S  0.0  2.0  0:02.69 kwin
 4564 detaos     20   0  198M  3136  2020 S  0.0  0.6  0:00.09 /usr/lib/libreoffice/pro
 3002 detaos     20   0  423M 17536  4120 S  0.0  3.5  0:02.31 /usr/bin/konsole
 2362 messagebu  20   0 32632  1048   548 S  0.0  0.2  0:00.15 /usr/bin/dbus-daemon --s
 2668 root       20   0  190M  1584   788 S  0.0  0.3  0:00.11 /usr/lib/policykit-1/pol
 2152 root       20   0  249M   568    84 S  0.0  0.1  0:00.13 /usr/sbin/rsyslogd
 2762 detaos     20   0  732M  5196  2968 S  0.0  1.0  0:00.66 kdeinit4: kded4 [kdeinit
 3400 detaos     20   0  130M 10296   348 S  0.0  2.0  0:00.42 php5 pts-core/phoronix-t
 3191 root       20   0 20340  1548   336 S  0.0  0.3  0:00.04 bash
 2984 detaos     20   0  676M  5008  2528 S  0.0  1.0  0:00.57 /usr/bin/krunner
 3148 detaos     20   0 20348  1504   292 S  0.0  0.3  0:00.04 /bin/bash
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

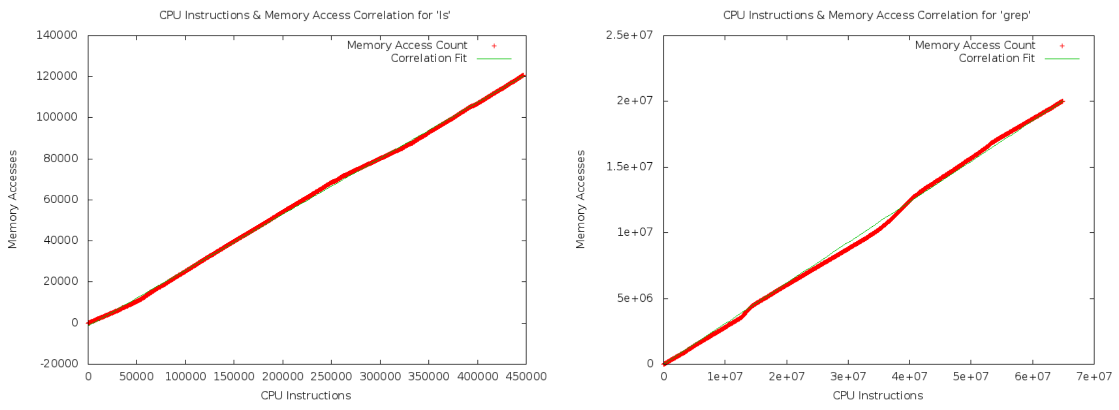
Figure 3.1: Screenshot of the data collection system running

3.2.1 Measurement Assumption

Ideally, performance would be measured by the number of machine instructions executed, not the amount of time passed. Unfortunately, there is no way to gather the number of instructions executed without running the system in something like Valgrind, which is unusably slow for any sufficiently interesting tests. Instead of machine instructions executed, we measure time passed. To support this substitution, we measured the number of machine instructions executed per time and found a strong linear correlation between them, see Figure 3.2. Therefore, we are comfortable using time as a replacement for machine instructions executed.

3.2.2 Limitations of Data Collection

Ideally, we would have collected the page use (read, write, modify) sequence along with information about every page in the system at the time of a page fault. Unfortunately, the kernel isn't notified of page events other than faulting, so the first isn't possible. And, due to the speed requirements of an IRQ handler, looping through even just the entire Inactive List during a page



(a) Valgrind output for ‘ls‘ (b) Valgrind output for ‘grep‘

Figure 3.2: Valgrind output showing a linear relationship between time and number of instructions executed

fault caused the system to crash. As such, we only gathered data about the page that caused the fault.

3.2.3 Recording Page Fault Events

To record the data, fields were added to two structs, `struct page` (see Appendix A.1) and `struct task_struct` (see Appendix A.2). `page` is the data type used to describe pages in memory. `task_struct` is the data type used to describe processes. Fields for the time a page's active bit was last cleared, time of last fault, and the index of a given page's previous fault's prior page that faulted were added to `page`. A field for time of last fault was added to `task_struct`. All other data were stored in static members of the page fault handler. The complete list of features collected can be found in Figure 3.3. The features we added are marked with "(MLVM)".

Features 14 and 16 may not be immediately obvious. See Figure 3.4 for a visualization of them. The sequence of blocks in the image represents the sequence of faulting pages. X is the current faulting page. Y_2 is the page that faulted prior the current faulting page, feature 14, herein referred to as ‘prior’. Y_1 is the page that faulted prior to the current pages' previous fault, feature 16, herein referred to as ‘prior-prior’.

3.3 Data Transformation and Analysis

The data collection procedures from the last section produced nearly 2GB of raw data which needed to be parsed and labeled before it could act as input for a machine learning algorithm. This section describes that transformation and some statistics about the data.

1. Time since last page fault (MLVM)
2. Page zone index
3. Page zone's inactive ratio
4. Page zone's spanned pages
5. Page zone's present pages
6. Page zone's managed pages
7. Previous faulting process' index (MLVM)
8. Current faulting process' index (MLVM)
9. Number of dirty pages belonging to the faulting process
10. Time since the faulting process last faulted (MLVM)
11. Faulting page's index
12. Time since faulting page's active bit was cleared (MLVM)
13. Time since faulting page last faulted (MLVM)
14. Index of the page that faulted prior to the faulting page's previous fault (MLVM)
15. Faulting page's reference counter
16. Index of the page that faulted prior to this page (MLVM)
17. The address that caused the fault
18. All page flags

Figure 3.3: Feature Vector List. Features we added to the kernel are annotated with "(MLVM)".

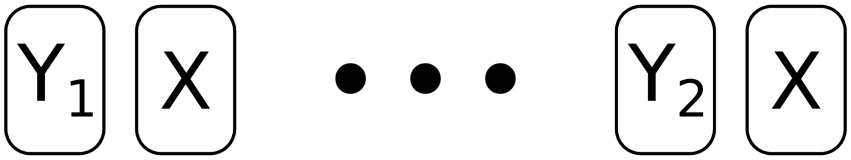


Figure 3.4: Visualization of the memory accesses pattern explaining features 14 and 16 from Figure 3.3.

3.3.1 Data Transformation and Labeling

Only a minor transformation had to be applied to the data before running it through Orange. All of the time values recorded in the raw data were given in absolute time, and had to be translated to relative time for the events.

To label the data, for each page fault event the remainder of the test's data was scanned to search for the next time that page faulted. The median, along with all other deciles for the next time a page faulted, was calculated (see Table 3.1). All faults below the 50 percent mark, that is, pages recalled from the backing store in less than approximately 127 seconds, were labeled as a bad decisions, while faults above the 50 percent mark, that is, pages that lived in the backing store for more than approximately 127 seconds, were labeled as good decisions.

Decile	Time until next page swap (nanoseconds)
10%	6624554
20%	214423654
30%	10025255994
40%	60660815059
50%	127280054223
60%	247219468813
70%	475254029795
80%	739048735114
90%	1752473234894

$P(\infty) : 0.507373$

Table 3.1: Deciles of time until next swap in nanoseconds and probability of a page not being recalled from the backing store, denoted here as $P(\infty)$.

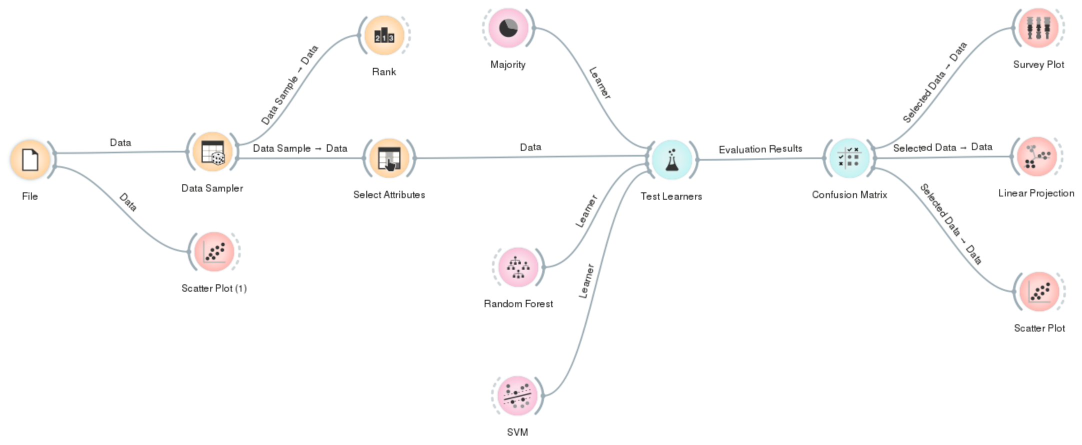


Figure 3.5: Screenshot of one of the Orange experimental environments we set up.

3.3.2 Machine Learning Software

In this research, we used the machine learning package Orange. It has a simple and intuitive graphical user interface, allowing the modern data mining scientist to accomplish most, if not all, of their work without having to write a single line of code, once the data is sufficiently transformed for input. See Figure 3.5 for a screenshot of one of the experimental environments we set up.

3.3.3 Calculating Baseline

A standard baseline would be the maximum-likelihood estimate (MLE); however, in some cases, this is artificially high due to the disproportionate presence of the class we're not looking for. We instead are using essentially the probability of selecting the item from the target class: call-all-X. For call-all-X, the precision is the proportion of the test set in the target class and recall is 1. In the cases where MLE was the probability of not-recalled, 50.7 percent, MLE was used.

3.4 Chapter Summary

In this chapter we presented the hypothesis for this research, explained the experimental setup for data gathering along with some limitations of our approach, presented the machine learning software package Orange, and the call-all-X method for calculating baseline, which will be used extensively in the following chapter. In the next chapter, we will present the machine learning experiments we conducted on various transformations of the data.

CHAPTER 4: Experimental Results

In this chapter, we present the sequence of experiments we performed on the data gathered in Chapter 3. Since the probability of a page never being recalled from the backing store was 50.7 percent, the initial task was to try to classify the data as either being recalled from the backing store at some point in the future, or not being recalled at all, we call this *recalled* or *not-recalled*, respectively. The data was randomly sampled to decrease computation time.

4.1 Phase 1: To Infinity and Beyond

Initially, we tried SVM (RBF kernel), naïve bayes, random forest, k-nearest neighbors, and CN2. The classification accuracy ranged from 0.55-0.60, with most classifiers only barely beating the MLE of 0.507. With such low scores, we looked to reduce the dimensionality, as it was suspected many of the features were adding noise. Using information gain, the page zone information, faulting address, and page flags were removed from the feature vector. Upon further reflection it was realized that the classifiers were trying to treat the process indexes as continuous variables when they are in fact metric-less identifiers. They were since transformed into a binary feature of whether or not the previously faulting process was the same as the currently faulting process. The page indices as absolute values are similarly meaningless by themselves, but due to spatial locality, pages with indexes close to each other could provide further insight. Therefore the index of the current faulting page was dropped and the prior and prior-prior indices were made relative to the current index. Shown in Figure 4.1 is the updated feature vector.

4.2 Phase 2: The Thin Red Line

With the reduced feature count, we retested the classifiers discussed in Section 4.1. The classification accuracy only improved slightly, with random forest remaining the best at 0.66. A linear projection (based on FreeViz [18]) of the reduced feature space showed evidence of a distinctly classifiable grouping, note the red line in Figure 4.2.

After performing further dimensional reduction based on the variance of the features for the data points on this distinct red line, the clumping turned out to be from the relation between the prior faulting page index and the prior-prior faulting page index. When plotted in a scatter plot,

1. Time since last page fault (MLVM)
2. Previous faulting process is the current faulting process (MLVM)
3. Number of dirty pages belonging to the faulting process
4. Time since the faulting process last faulted (MLVM)
5. Time since faulting page's active bit was cleared (MLVM)
6. Time since faulting page last faulted (MLVM)
7. Faulting page's reference counter
8. Distance of the index of the page that faulted prior to the faulting page's previous fault from the current faulting page (MLVM)
9. Distance of the index of the page that faulted prior to this page from this page (MLVM)

Figure 4.1: Updated Feature Vector List. Features we added to the kernel are annotated with "(MLVM)".

see Figure 4.3, it was obvious that the equation of the red line was $y = x$, i.e., when the prior faulting page index was the prior-prior faulting page index. What we were seeing was the result of regularly repeated code paths, which is consistent with the idea of temporal locality.

4.3 Phase 3: Just Another Case of History Repeating

We split the data according to whether the prior faulting page was also the prior-prior faulting page to process the two groups separately. Taking the data group where they are the same first, we reran the classifiers to look for not-recalled. Random forest again came out on top. With the newly limited data set, it achieved an accuracy of 0.93 with an F-Score of 0.81. Precision and recall were 0.82 and 0.81, respectively. Call-all-not-recalled's F-Score and thus the baseline was 0.30. SVM achieved an F-Score of 0.66. The precision and recall were 0.80 and 0.57, respectively.

The predominately deciding feature turned out to be the process' number of dirty pages with an information gain of 0.307. The other major factors were the time since the page had last faulted and the time since the page's active bit had been cleared with information gains of 0.237 and 0.219, respectively. See Table 4.1 for the complete list of the features' information gain.

As is demonstrated in Figure 4.4, the data when prior is the same as prior-prior is clearly separable. In the next set of experiments we turn to predicting good pages to throw out when prior is different from prior-prior.

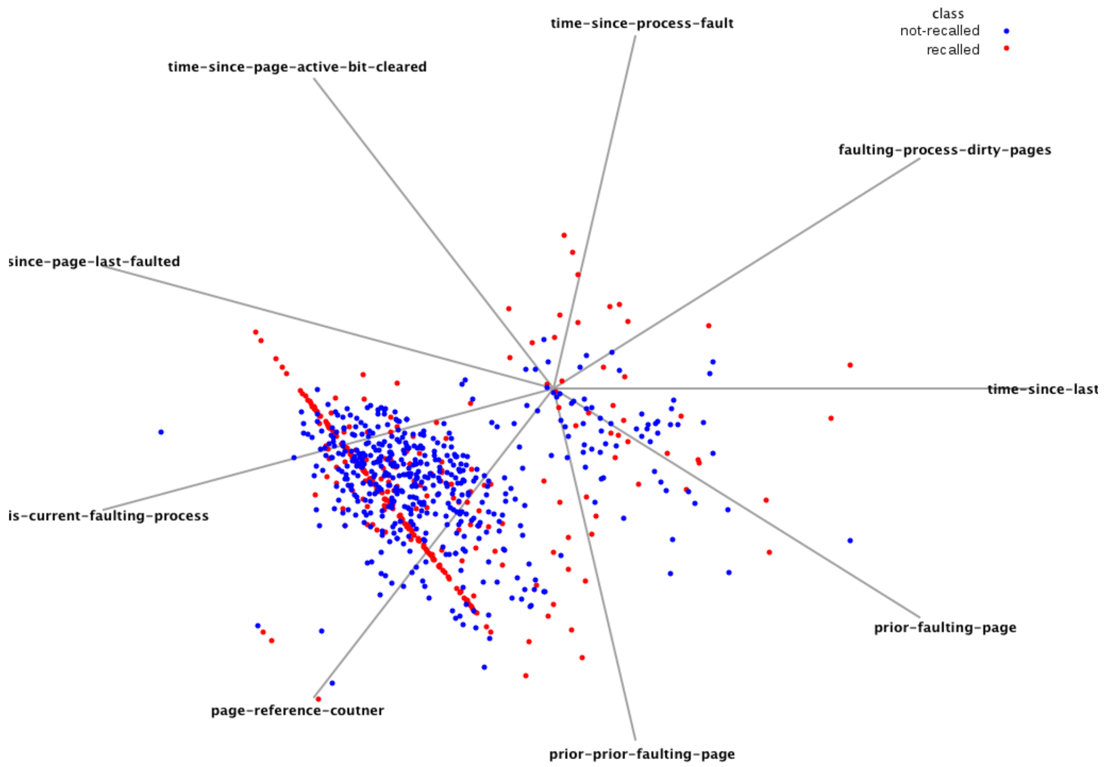


Figure 4.2: Linear projection (based on FreeViz [18]) of updated feature vector. Note the distinct red line in the dots in the lower left corner of the projection. All of the dots on this line are those pages who's prior faulting page was the same as its prior-prior faulting page, 10 percent of which will never be swapped back in.

4.4 Phase 4: The Only Constant is Change

Classifying the remaining data, i.e., when the prior faulting page index was not equal to the prior-prior faulting page index proved to be far more of a challenge than classifying when prior is the same as prior-prior. MLE's accuracy was 0.52, so the data was more evenly split. The baseline F-Score was 0.69. This time, SVM beat out random forest, but neither classifier managed to beat baseline. SVM achieved an F-Score of .66 while Random Forest only managed 0.63.

As is clearly demonstrated in Figure 4.5 and backed up by the lack of information gain in Table 4.2, there are no discernible groupings in the data, which explains the poor F-Scores.

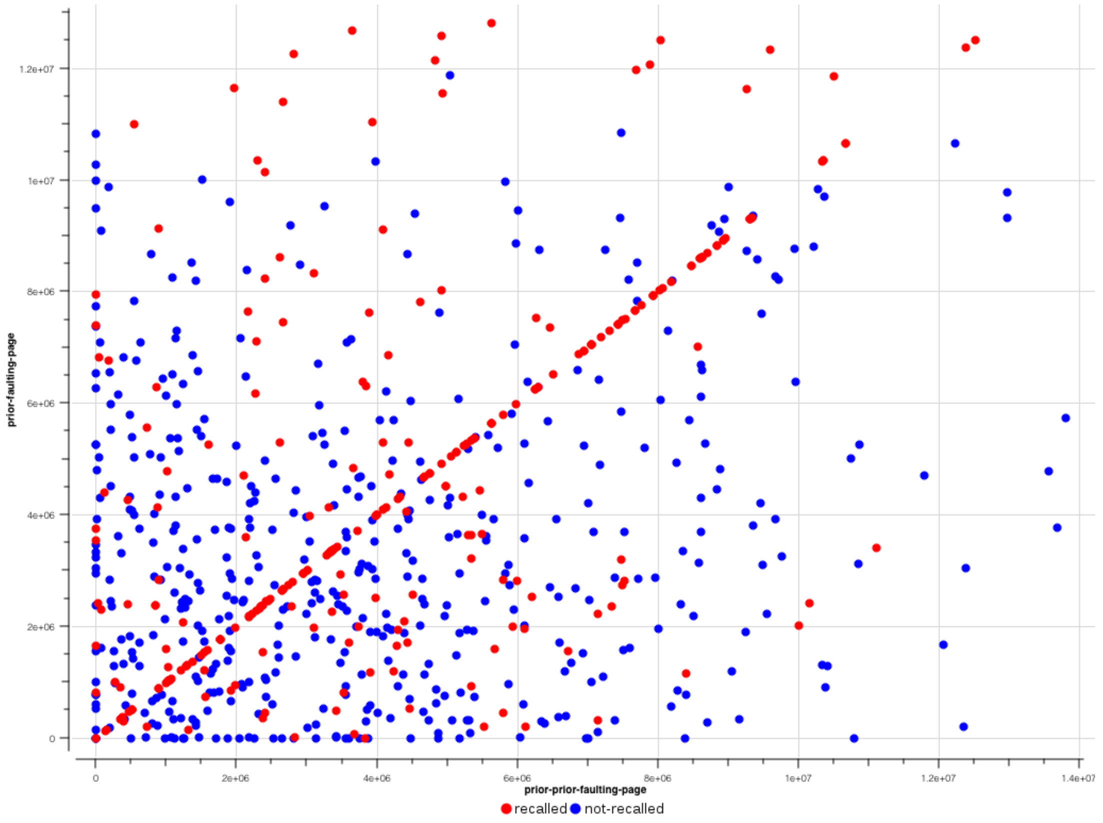


Figure 4.3: Scatter plot of prior faulting page index against prior-prior faulting page index showing the distinct red line along $y = x$ indicating pages where the prior faulting page was the same as the prior-prior faulting page.

4.5 Phase 5: The Good, the Bad, and the Ugly

Supposing for a moment that we could successfully classify a page as not-recalled or recalled, we resplit the data and worked only with pages previously labeled as recalled, i.e., only those pages which were recalled from the backing store. The goal was to see if we could distinguish between pages falling above or bellow the 50 percent threshold, calling pages which fell bellow the threshold bad page out decisions and those above the threshold as being good page out decisions. From our previous experiments, we added a feature to the feature vector: a binary feature noting whether or not prior is the same as prior-prior.

Since the data were split exactly in half, MLE was 0.50 and thus used as baseline, achieving an F-Score of 0.691. Both SVM and random forest were able to beat baseline, achieving F-scores of 0.754 and 0.744, respectively.

Attribute	Information Gain
Faulting process dirty pages	0.307
Time since page last faulted	0.237
Time since page's active bit cleared	0.219
Prior prior faulting page	0.012
Prior faulting page	0.012
Page reference counter	0.004
Time since process fault	0.001
Time since last fault	0.001
Previous is current faulting process	0.000

Table 4.1: Information Gain for the updated feature vector for the data when prior is the same as prior-prior showing strong discernability among the top features.

Attribute	Information Gain
Time since page last faulted	0.008
Faulting process dirty pages	0.006
Time since process fault	0.001
Previous is current faulting process	0.001
Time since last fault	0.001
Page reference counter	0.001
Prior prior faulting page	0.001
Prior faulting page	0.001
Time since page's active bit cleared	0.000

Table 4.2: Information gain for the updated feature vector of prior is different from prior-prior data showing far less discernability between the features.

4.6 Phase 6: Back to Reality

Since we were not able to determine not-recalled vs recalled well, we then decided to test classifying above or bellow the 50 percent threshold using all of the data. For this data set, MLE for the class we weren't looking for was 0.75, so call-all-recalled was used as baseline, which achieved an F-Score of 0.385. SVM was only barely able to beat baseline with an F-Score of 0.395, while Random Forest was able to get 0.477.

Looking at the information gain, see Table 4.3 for the complete table, it appeared as though some of the features were still adding noise. We took the top four features and saw a slight improvement in SVM's F-Score with 0.402, while random forest fell to 0.471.

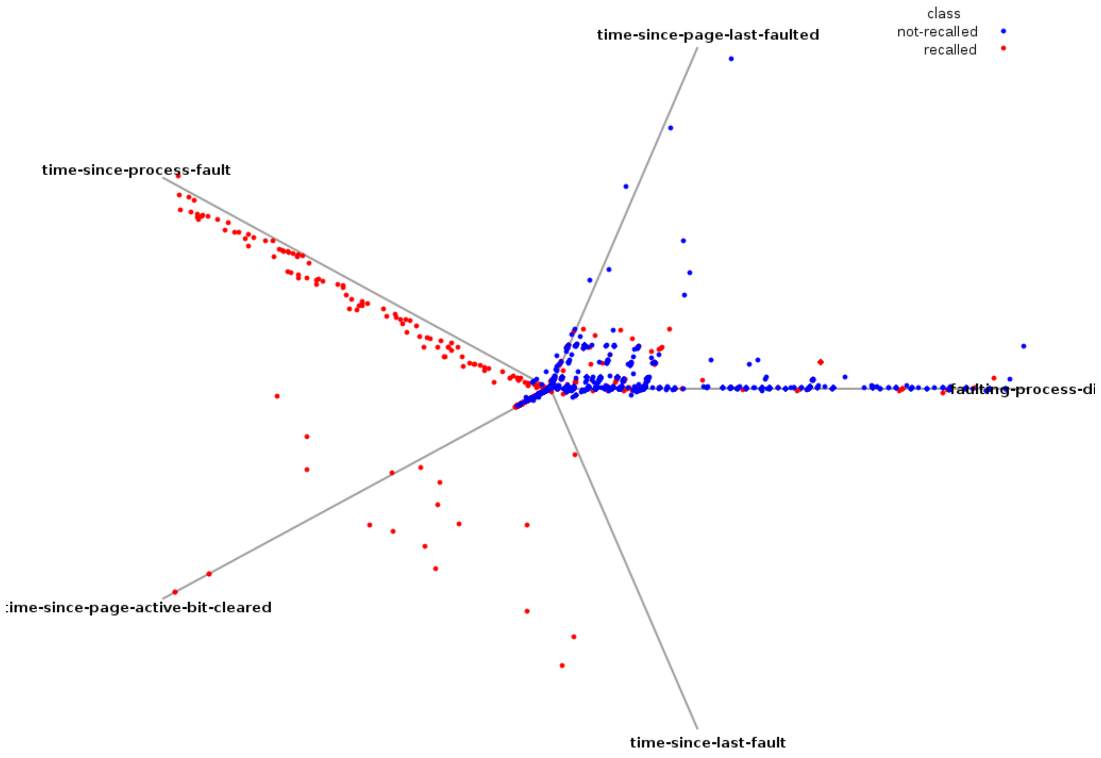


Figure 4.4: Linear projection (based on FreeViz [18]) of when prior is the same as prior-prior data showing how highly separable the data is.

4.7 Phase 7: Taking the Log

While looking at the survey plot for 50 percent data split, we noticed that the time related data was showing a *step-exponential* shape (see Figure 4.6). Since we used SVM with normalization, it made sense to take the log of all time values to increase the separation of the lower values. The data split for this set was the same as the last set with an MLE of 0.75 of the non-target class, so call-all-recalled was again used for baseline with the same F-Score of 0.385. Both random forest and SVM saw slight improvements, achieving F-Scores of 0.487 and 0.403, respectively.

Noting similar patterns in the linear projection and information gain as we saw in Phase 6, we repeated the feature-space reduction. In contrast to Phase 6 where feature-space reduction caused a drop for random forest, both random forest and SVM saw marginal increases to 0.505 and 0.406 respectively.

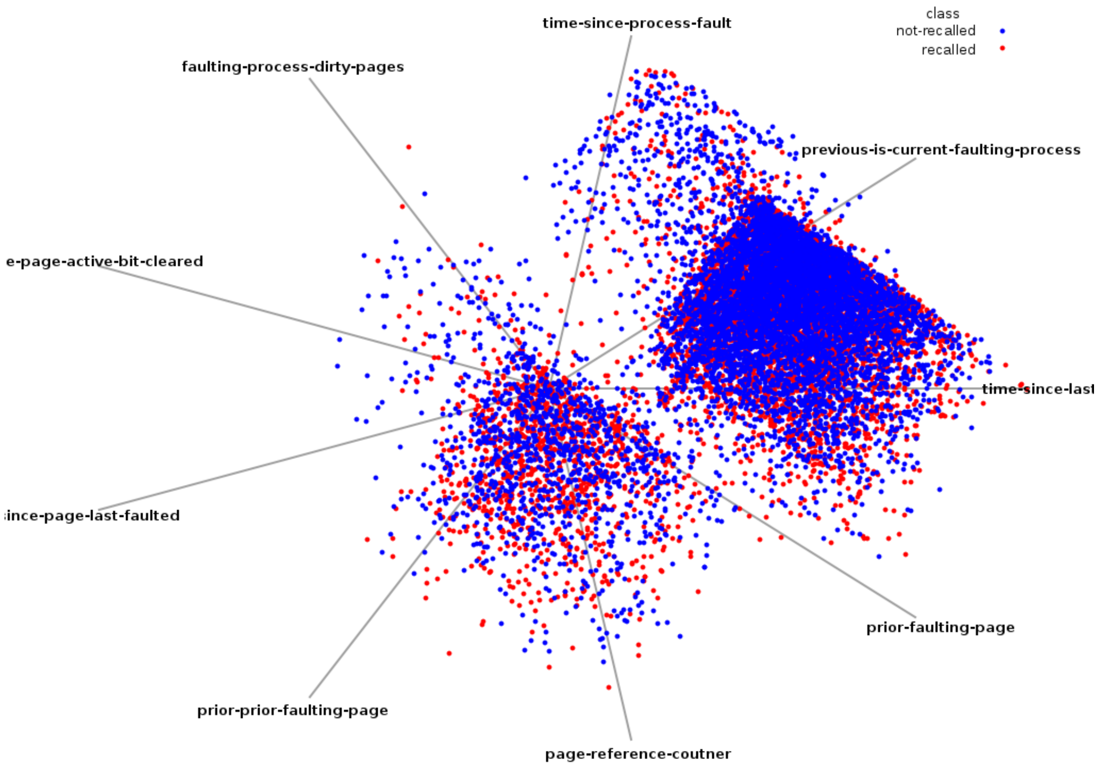


Figure 4.5: Linear projection of prior is different from prior-prior data showing how not separable the data is.

4.8 Phase 8: Low Hanging Fruit

Taking one last look at the data, we decided to see if we could classify the worst of the pages, those that were below the 20 percent mark, ie pages that lived in the backing store for less than a second. With an MLE of only 0.10, call-all-recalled was again used for baseline, with an F-Score of 0.124. Both Random Forest and SVM performed well, achieving F-scores of 0.786 and 0.644, respectively. For Random Forest, the precision and recall were 0.818 and 0.758, respectively. For SVM, precision and recall were 0.693 and 0.600, respectively. Hoping for another marginal increase, we applied the same feature-space reduction from Phase 6 and 7 only to be disappointed as both fell to 0.770 and 0.640, respectively.

4.9 Chapter Summary

In this chapter we presented the findings from the various methods we applied to split and transform the data. In most cases we were not able to do appreciably better than MLE, but in

Attribute	Information Gain
prior is the same as prior-prior	0.084
Faulting process dirty pages	0.059
Time since page last faulted	0.039
Time since page's active bit cleared	0.009
Page reference counter	0.003
Time since last fault	0.002
Time since process fault	0.001
Prior faulting page	0.001
Prior-prior faulting page	0.001
Previous is current faulting process	0.000

Table 4.3: Information gain for the feature vector of the Phase 6 data showing moderate separability.

two cases we did very well. In the first case, when the prior faulting page was the same as the prior-prior faulting page, we succeeded in classifying those pages that would not be recalled from the backing store. In the second case, we succeeded in classifying those pages that fell below the 20 percent mark, i.e., the pages that lived in the backing store for less than one second, making them very bad page out decisions.

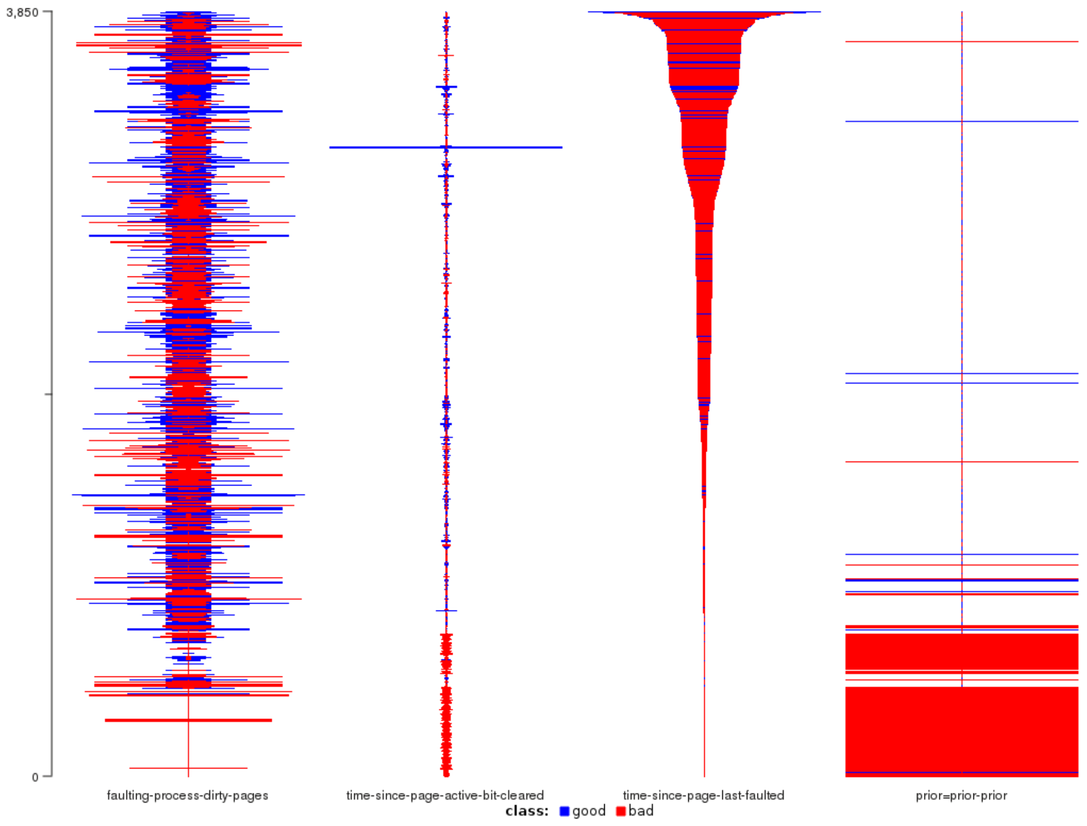


Figure 4.6: Survey plot of features from the Phase 6 data split sorted by time since page last faulted showing a step-exponential shape. All columns are sorted according to the order resulting from sorting time since page last faulted. The lack of a discernible patterns in the other three columns demonstrates the lack of a simple correlation among the features.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5: Conclusions and Recommendations

In this thesis, we presented a method of modifying the Linux kernel for data collection about page-out events. This method was used to build a data set useful for input into machine learning algorithms. From the data set, we discovered two cases in which improvements to Linux's page replacement algorithm can be made.

As we discovered in Section 3.3, over half of the pages swapped out were never needed again. As such, Linux's 2nd Chance 2Q Clock PRA (see Section 2.1.3) is an effective page replacement algorithm. As shown in Figure 5.1, $TF(p)$ the time until the next fault page p has an exponential shape, when sorted by time. Linux's page fault algorithm is not ideal; too many pages have a short time until next fault. Ideally, that graph would be more logarithmic than an exponential. However, the graph shows only half of the data; the other half never faulting after the initial load, so $TF(p)$ is infinite.

5.1 Contributions

Though many of the tests described in Chapter 4 failed to, or only barely beat, the baseline for the given test, performance was notable in two key cases. First, demonstrating temporal locality, pages where the prior faulting pages was the same as the prior-prior faulting page were easily classified as not-recalled vs. recalled. Second, the pages from the worst page-out decisions, those falling below the 20 percent decile who lived in the backing store for less than a second, were successfully classified against those who lived in the backing store longer. Exploiting these two cases in the page replacement algorithm might provide significant system performance gains by decreasing the number of bad page out decisions.

5.1.1 The Thin Red Line

In the workload we used for data collection, pages were found on the red line seen in Figure 4.2 indicating that the prior faulting page was the same as the prior-prior faulting page 10.0 percent of the time. Since the line itself is a product of the way we program with loops and regularly repeated code paths, i.e., temporal locality, some pages should fall on it no matter the workload. The only thing that will vary is the percentage of pages found on the line.

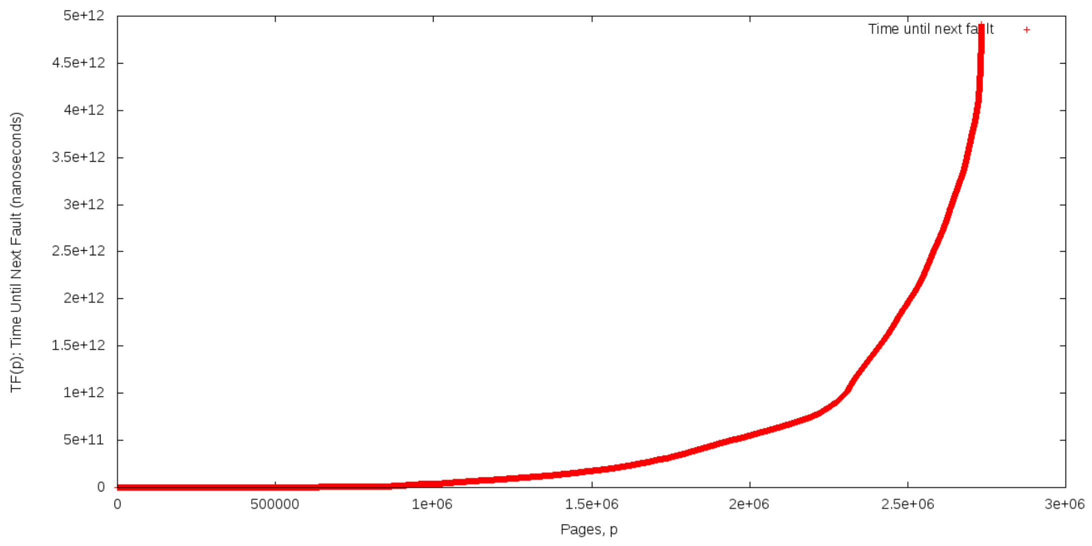


Figure 5.1: $TF(p)$: Time until next fault of pages, sorted by time

For our workload, pages found on the line were recalled from the backing store 90 percent of the time. We were reasonably good at selecting the remaining 10 percent of the pages. For Random Forest, the precision and recall were 0.819 and 0.805, respectively. For SVM, the precision and recall were 0.800 and 0.566, respectively. These pages are good candidates for eviction.

There is still an issue however for the candidate pages chosen above. If we happen to be wrong, and the page chosen has a non-infinite time until next fault it is likely to have a very short time until next fault (see Figure 5.2). The following section describes a method for dealing with this problem.

5.1.2 Below the 20 Percent Decile

Pages that fall below the 20 percent decile are the worst candidates for paging out. They live in the backing store for less than one second before being recalled into RAM. Luckily, we were able to build a classifier to determine if the candidate page is in this set. For random forest, the precision and recall were .818 and .758, respectively. For SVM, precision and recall were 0.693 and 0.600, respectively. Compare this to a baseline of 0.124.

5.2 Future Work

This thesis has laid the ground work for a more mature data collection environment. The first thing that should be done is use existing infrastructure to conduct more tests under different workloads and look for artifacts similar to *the thin red line*. Additionally, it may be worth-

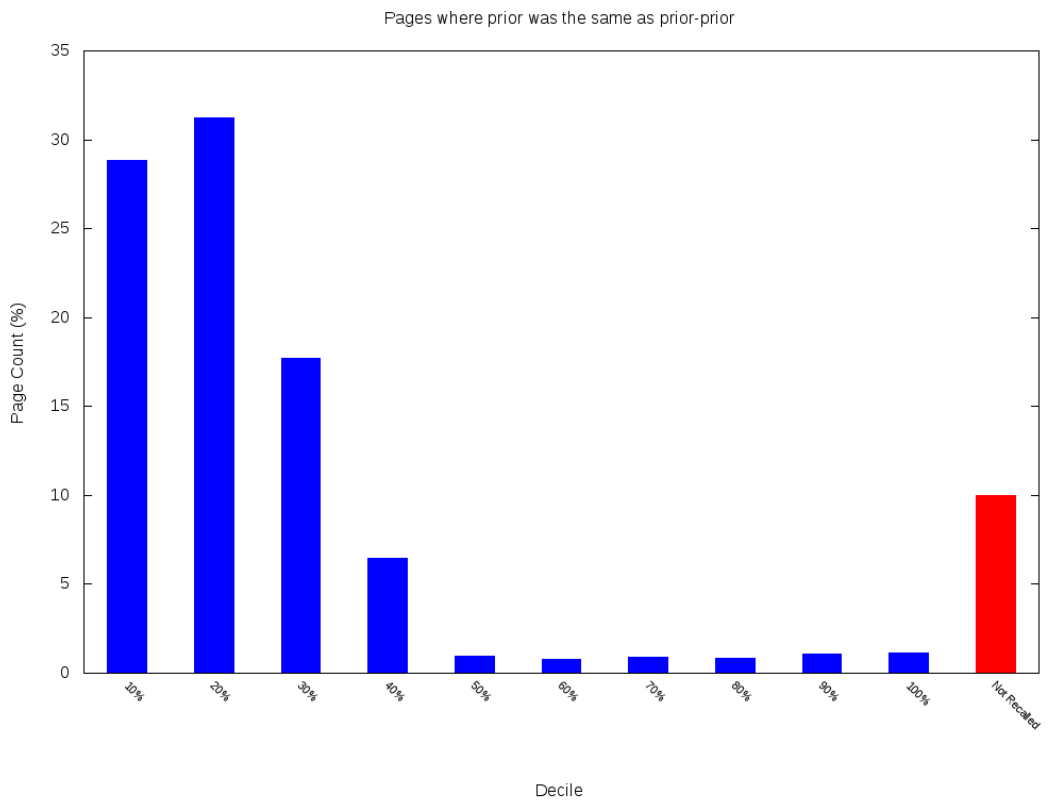


Figure 5.2: Histogram showing the distribution of pages on the red line among the deciles. Pages in the first two bars are swapped back in within 0.25 seconds. Pages in the first four bars are swapped back in in under a minute.

while to investigate the graphs in Figure 5.3 and Figure 5.4, which are plotted from the data from Phase 7. Other relationship may be discovered here that could lead to further insight into possible improvements for Linux's page replacement algorithm.

5.2.1 Machine Learning Swap Algorithm Implementation

The ultimate goal is to implement and test a page replacement algorithm based on this and future research. The normal page replacement algorithm would identify a candidate for eviction. It would then pass that page to our system built to be classified as a good or bad candidate for eviction. See Figure 5.5 for the pseudo-code algorithm. After implementation, performance tests would need to be conducted to tune the counter threshold (set to three initially) to see if the new algorithm could outperform the current one. The above tests must be designed to balance actual system performance against the cost of producing the ideal page replacement algorithm.

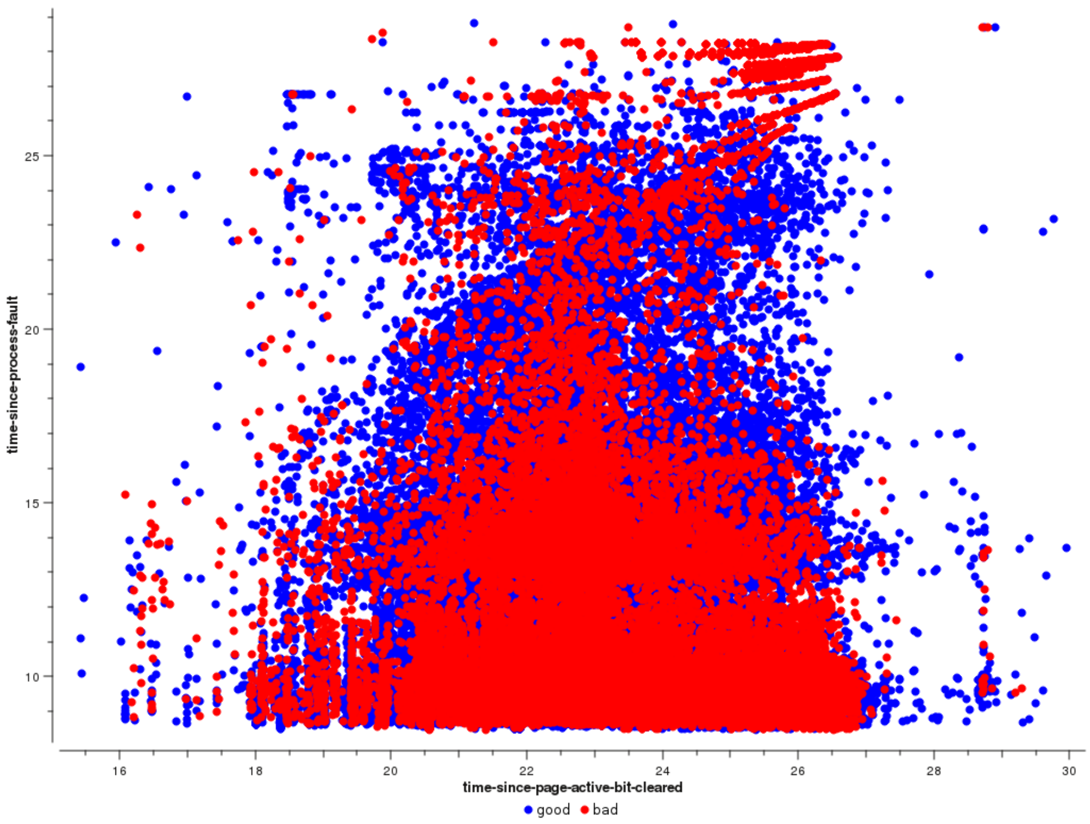


Figure 5.3: Scatter plot of the time since a page's active bit was cleared vs the time since its containing process had last faulted.

5.2.2 The Other Side of the Thin Red Line

Supposing that implementing a machine learning algorithm in the kernel is slower than the current implementation, despite making better page-out decisions, a simpler approach can be found in the data in Figure 5.2. The vast majority of pages found on the thin red line fall below the 40 percent mark and comprise the worst page-out decisions. Simply withholding all pages with prior equal to prior-prior could provide a system speedup and would be far easier to implement. All that would be required are the additions to track prior and prior-prior in the page struct, which would get updated in the page fault handler, with a simple test for equality during the page-out process.

5.3 Conclusion

In this thesis, we modified the Linux kernel for data mining the page-out decisions made by Linux's page replacement algorithm. We transformed that data for input into machine learning algorithms and identified two cases where improvements could be made to Linux's PRA. The

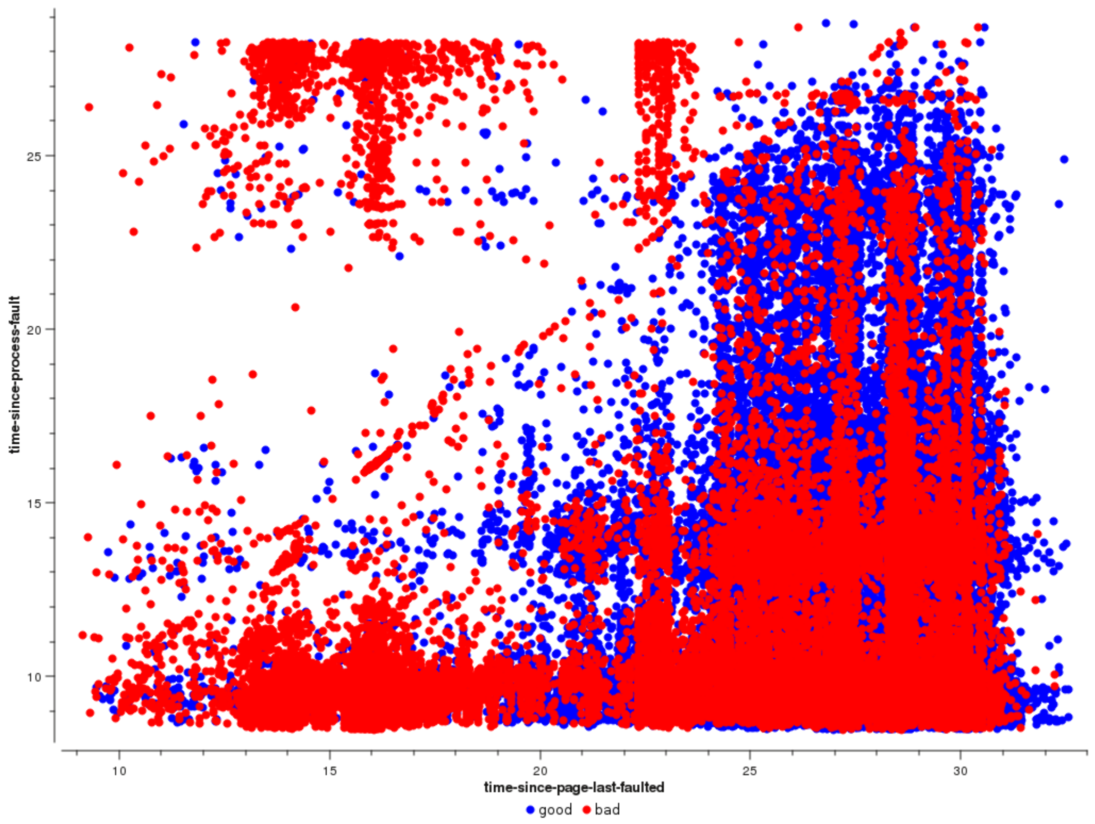


Figure 5.4: Scatter plot of the time since the page had last faulted vs the time since its containing process had last faulted.

1. Set counter to zero
2. Page replacement algorithm (PRA) selects candidate for eviction
3. PRA queries trained classifier to judge candidate page
4. If page is not-recalled proceed with eviction
5. Else skip page, increment counter
6. If counter < three goto 2.
7. Else evict page anyways, goto 1.

Figure 5.5: Algorithmic addition to Linux's current page replacement algorithm

first case being when the prior faulting page was the same as the prior-prior faulting page. It was noted that 90 percent of the pages with prior equal to prior-prior would be recalled, the vast majority of which being the worst candidates for eviction and suggested an improvement to Linux's PRA based on keeping all pages with prior equal to prior-prior in RAM. The second case being when the page lived in the backing store for less than one second before being recalled into RAM and suggested keeping those pages in RAM in favour of pages with longer expected stays in the backing store. While implementation testing is clearly needed to determine the impacts of these enhancements, this thesis has identified a couple avenues for improving the performance of the billions of devices that run Linux.

APPENDIX A:
Data Structure Modifications

A.1 /include/linux/mm_types.h

```
struct page {
    ...

    /* MLVM additions */
    struct timespec active_bit_cleared;
    struct timespec last_fault_time;
    unsigned long previous_faults_prior_page_faulted;
};
```

A.2 /include/linux/sched.h

```
struct task_struct {
    ...

    /* MLVM additions */
    struct timespec last_fault_time;
};
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B:
Bash Scripts

B.1 run_test.sh

```
#! /bin/bash

cd mlvm_log_module
make
cd ..

i=1

while true
do
    echo "Test run: $i"

    cd mlvm_log_module
    insmod ./mlvm_log_module.ko
    cd ..

    su - [username] -c ./run_phoronix.sh &

    su - [username] -c ./induce_memory_pressure.sh > /dev/null

    rmmod mlvm_log_module

    i=$(($i+1))
done
```

B.2 run_phoronix.sh

```
#! /bin/bash
```

```
phoronix-test-suite interactive << ARGS
3
n
7
ARGS
```

B.3 induce_memory_pressure.sh

```
#! /bin/bash

websites=(
www.cnn.com
www.foxnews.com
www.imgur.com
www.reddit.com
www.thechive.com
www.knowyourmeme.com
www.mashable.com
www.wsj.com
)

# --norestore flag used to prevent attempting to recover the file that was
# not saved when the program was terminated by the kill command
office_apps=(
"localc --norestore"
"lodraw --norestore"
"loimpress --norestore"
"lomath --norestore"
"lowriter --norestore"
)

actions=(
internet
office
)

function rando_sleep {
    sleep $(( $1 + ($RANDOM % $2) ))
}

function internet {
    #TODO clear browser cache
    for i in { 1..$(( 2 + ($RANDOM % 10) )) }
    do
        chromium ${websites[ $(($RANDOM % ${#websites[@]})) ]} &> /dev/null &
        rando_sleep 5 5
    done
    killall chromium
    while $(ps -C chromium > /dev/null)
    do
        sleep 10
    done
}

function office {
    ${office_apps[ $(($RANDOM % ${#office_apps[@]})) ]} &
    rando_sleep 10 10
    killall soffice.bin
    while $(ps -C soffice.bin > /dev/null)
    do
        sleep 10
    done
}

while $(ps -C phoronix-test-suite > /dev/null)
do
    ${actions[ $(($RANDOM % ${#actions[@]})) ]}
    if $(ps -C x-www-browser > /dev/null)
    then
        killall x-www-browser
    fi
    if $(ps -C iceweasel > /dev/null)
    then
        killall iceweasel
    fi
    rando_sleep 5 5
done
```

REFERENCES

[1] L. A. Bélády, “A study of replacement algorithms for a virtual-storage computer,” *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, Feb. 1966.

[2] P. J. Denning, “The working set model for program behavior,” *Communications of the ACM*, vol. 11, no. 5, pp. 323–333, May 1968.

[3] L. A. Bélády et al., “An anomaly in space-time characteristics of certain programs running in a paging machine,” *Communications of the ACM*, vol. 12, no. 6, pp. 349–353, Jun. 1969.

[4] A. McMenamin, “Applying working set heuristics to the Linux kernel,” M.S. thesis, Dept. Computer Science and Information Systems, Univ. London, London, UK, 2011.

[5] N. Megiddo and D. S. Modha, “ARC: A self-tuning, low overhead replacement cache,” in *Proc. FAST 2003: 2nd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 2003, pp. 115–130.

[6] S. Bansal and D. S. Modha, “CAR: Clock with adaptive replacement,” in *Proc. FAST 2004: 3rd USENIX Conference on File and Storage Technologies*, San Francisco, CA, 2004, pp. 187–200.

[7] W. Stallings, “Virtual memory,” in *Operating Systems: Internals and Design Principles*, 7th ed. Upper Saddle River: Pearson/Prentice Hall, 2009, ch. 8, sec. 2, pp. 370–374.

[8] R. W. Cart and J. L. Hennessy, “WSClock — A simple and effective algorithm for virtual memory management,” M.S. thesis, Dept. Computer Science, Stanford Univ., Stanford, CA, 1981.

[9] A. S. Tanenbaum, “The least recently used (LRU) page replacement algorithm,” in *Modern Operating Systems*, 3rd ed. Upper Saddle River: Prentice Hall, 2001, ch. 4, sec. 4, pp. 218–220.

[10] X. Ge et al., “Local context matching for page replacement.” M.S. thesis, Dept. Computer Science, Univ. California, Irvine, 1999.

[11] T. Johnson and D. Shasha, “2Q: A low overhead high performance buffer management replacement algorithm”, M.S. thesis, Dept. Computer Science, Univ. Florida, Gainesville, 1994.

[12] M. Gorman, “Page frame reclamation,” in *Understanding the Linux Virtual Memory Manager*, Upper Saddle River: Prentice Hall, 2004, ch. 10, sec. 1, pp. 164–165.

[13] C. Cortes and V. Vapnik, “Support-Vector Networks,” *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep. 1995.

[14] L. Breiman, “Random Forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001.

[15] R. Stuart, “Quantifying Uncertainty,” in *Artificial Intelligence: A Modern Approach*, 2nd ed. Upper Saddle River: Pearson Education, 2003, ch. 13, sec. 5, pp. 495–497.

[16] D. Bremner et al., “Output-sensitive algorithms for computing nearest-neighbour decision boundaries,” *Discrete & Computational Geometry*, vol. 33, no. 4, pp. 593–604, Apr. 2005.

[17] P. Clark and T. Niblett, “The CN2 induction algorithm,” *Machine Learning*, vol. 3, no. 4, pp. 261–283, Mar. 1989.

[18] J. Demsar et al., “FreeViz — An intelligent visualization approach for class-labeled multidimensional data sets,” in *Proc. Intelligent Data Analysis in Medicine and Pharmacology*, Aberdeen, UK, 2005, pp. 61–66.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia
2. Dudly Knox Library
   Naval Postgraduate School
   Monterey, California